Advanced XML / Data on the Web

Lecture 7

Lars Birkedal [birkedal@it-c.dk]

The IT University of Copenhagen

Outline of this lecture

- Homework
- Types III: Regular Expression Types for XML
  - XDuce
- Readings:
  - Hosoya, et. al.: XDuce: A Typed XML Processing Language
  - Hosoya et. al.: Regular Expression Types for XML
  - Optional: Hosoya et. al.: Regular Expression Pattern Matching for XML.

Homework

- bal

XDuce

- Goal: static type checking of programs for XML processing
- Inspired by functional programming languages
- Novel features:
  - regular expression types
  - powerful notion of subtyping
  - regular expression pattern matching

Regular Expression Types

- XDuce's values are XML documents
- XDuce value:

```
val mybook = addrbook[
    name=["Haruo Hosoya"],
    addr=["Tokyo"],
    name=["ABC"],
    addr=["Def"],
    tel=["123-456-789"],
    name=["Benli Pierce"],
    addr=["Philadelphia"]]
```

Regular Expression Types

- Corresponding XML document

```
<addrbook>
  <name>Haruo Hosoya</name>
  <addr>Tokyo</addr>
  <name>ABC</name>
  <addr>Def</addr>
  <tel>123-456-789</tel>
  <name>Benli Pierce</name>
  <addr>Philadelphia</addr>
</addrbook>
```

Regular Expression Types

- Types:

```
type Addrbook = addrbook[(Name,Addr,Tel?)]
type Name = name[String]
type Addr = addr[String]
type Tel  = tel[String]
```

- Type formers: `|`, `*`, `?`, `,`
- Correspond to obvious DTD

Subtyping

- All values in XDuce are sequences!
- `()` — the empty sequence
- `tel["123"]` — one element sequence
- `tel["123"],tel["456"]` — two element sequence
- Comma `,` is used in types for concatenation of sequences
- Example: type `(Name,Tel*,Addr)` contains `name["a"],tel["1"],tel["2"],addr["b"]`
- Comma `,` is associative.
- Subtyping = inclusion of sets denoted by types (formal def'n later)

Subtyping

- We now show that `mybook` has type `Addrbook`. Observe:

```
Name,Addr <: Name,Addr,Tel?
Name,Addr,Tel <: Name,Addr,Tel?
T,T,T <: T*
```

- so

```
(Name,Addr),(Name,Addr,Tel),(Name,Addr)
  <: (Name,Addr,Tel?)*
```

- so (using that comma is assoc)

```
Name,Addr,Name,Addr,Tel,Name,Addr
  <: (Name,Addr,Tel?)*
```

Subtyping

- hence

```
addrbook[Name,Addr,Name,Addr,Tel,Name,Addr]
  <: addrbook[(Name,Addr,Tel?)*]
```

- since `mybook` clearly has type on the left, it also has type on the right.

Union Types

- Basic subtyping relationship for union types:

```
Name <: Name | Tel
Tel  <: Name | Tel
```

- “forget ordering” subtyping:

```
(Name,Addr)*, (Name,Tel)*
  <: ((Name,Addr) | (Name,Tel))*
```

- useful for evolving databases
- “distributivity” subtyping:

```
(Name,Addr) | (Name,Tel)
= Name, (Addr|Tel)
```

(where A=B means A<:B and B<:A)

Reg Exp Pattern Matching

```
fun mkTelList : (Name,Addr,Tel?)*
                -> (Name,Tel)* =
  name[n:String], addr[a:String],
    tel[t:String], rest:(Name,Addr,Tel?)*
    => name[n], tel[t], mkTelList(rest)
| name[n:String], addr[a:String],
      rest:(Name,Addr,Tel?)*
    => mkTelList(rest)
| ()
    => ()
```

Reg Exp Pattern Matching

```
fun firstTriple : (Name,Addr,Tel?)*
                  -> (Name,Addr,Tel?)
  ps:(Name,Addr)*, t:(Name,Addr,Tel),
    rest:(Name,Addr,Tel?)*
    => t
| whole:(Name,Addr,Tel?)*
    => whole
```

Note:

- first match, longest match policy
- (this kind of matching for *variable* length sequences is beyond ML pattern matching)

Reg Exp Pattern Matching

- Check for exhaustiveness (all cases covered) done by checking subtyping relation:

```
(Name,Addr,Tel?)*
<: name[String], addr[String],
     tel[String], (Name,Addr,Tel?)*
   | name[String], addr[String],
     (Name,Addr,Tel?)*
   | ()
```

XDuce Precisely

Done on the blackboard, following Appendix in XDuce: A Typed XML Processing Language.